Project Stage 12: Proper word searching

Work in *self-assigned pairs* on this stage.

You will improve the search capability of the Bible Reader by searching on ***whole words*** and ***phrases***. For instance, searching for:

```
"Son of God" man
```

should return all verses that contain the exact phrase *Son of God* and also contain the word *man* (which does *not* include verses containing "many", for instance (unless they also contain "man" as well)).

As before, you should display the results for all three versions (or as many as are open) and include a verse if there is a match in *any* of the versions.

You will do this by first implementing a ***Concordance*** class. You will then implement a few additional methods in the model and make appropriate changes to the remainder of your classes.

In case you are not familiar with the concept, a concordance is simply a book that allows you to look up words and it lists every verse that contains that word. In other words, you want to associate with each word a collection of references. A concordance is associated with a particular version of the Bible, so if you have three Bibles, you will have three concordances.

1. You will probably want to modify ***BibleFactory.createBible*** so that it returns an instance of ***TreeMapBible*** instead of ***ArrayListBible*** since the former will probably perform better overall. But make sure that the ***TreeMapBible*** passes all of the tests from Stage 11.
2. Implement the ***Concordance*** class. The bulk of the work should be done in the constructor of the Concordance class. In other words, you should not perform a search every time a request is made. Instead, you should have most of the results already prepared. In particular, ***getReferencesContaining*** should be a few lines long, and only because you need to do some error checking. ***getReferencesContainingAll*** will be a bit more work to get correct and efficient.
3. In your model, you need to construct and store a concordance for each version of the Bible that is added to it. You also need to implement three methods: ***getReferencesContainingWord***, ***getReferencesContainingAllWords***, and ***getReferencesContainingAllWordsAndPhrases*** (in that order). Note that these methods will use methods from the Concordance class, but they need to do the work of combining results.
4. Instead of going into greater detail about how to implement these methods, I want you to look at the tests and experiment with my solution. As you look at the tests, keep in mind that the quotes need to be escaped in the Java code, so instead of seeing "son of god", you will see \"son of god\". If you copy and paste this into your (or my) application, don't forget to remove the slashes.
5. Your model now has four possible methods to call to perform a word search. Your ***BibleReaderApp*** should call the best of these methods that you are able to properly implement.
6. Update your application so that everything works as before, but with the search results now based on whole-word searching.
7. Make sure that searched words are still highlighted properly.

8. Do not worry about only highlighting matches to exact phrases (e.g. if you search for "Son of God" (with the quotes), it should highlight that phrase, but might also highlight each of those words that occur elsewhere in the verse). It is a little tricky to get this just right.
9. Copy and run the Stage 12 tests. It is a little more difficult to pass *all* of the tests this time, but do your best. The tests are numbered in the order you should attempt to pass them.
10. Create **MyGrade_P12.txt** and fill in your actual time spent and expected grade and include a brief justification of your expected grade.
11. Zip up your **src** directory and submit it and **MyGrade_P12.txt** using Handin under assignment **235-P12**.
12. Grades will be based on passing the tests, the total time the tests take to run, the implementation of the Concordance constructor and the 5 search methods described above, and a properly implemented and functioning GUI.

Hints/Tips

- If you run out of heap space when you run the tests, you can increase the heap size in Eclipse as follows:
    - Select Run->Run Configurations...
    - In the left side of the window, find the JUnit heading.
    - Look under the JUnit heading for the test you are running and click on it (It might be the class name or the package name, depending on whether you run them one at a time or all together).
    - In the right side of the window, make sure the **name** field has the package/class you expect.
    - Click on the Arguments tab
    - In the box under VM Arguments (NOT Program arguments), put: **-Xmx256M**

  This will increase the maximum heap size to 256MB. You can change that to 512 or larger if you need to. If you run the tests separately, you will need to do this for each test.
- If a word occurs twice in a verse, you should only list it once in your concordance. It will hurt your overall performance, and possibly the results, if you do this incorrectly. (Not that I know from personal experience or anything.) This may suggest a particular type of collection and/or that you need to be careful as you construct the concordance.
- Let me be a little more direct with a hint, but still not giving you everything: If I ask the concordance "Hey, what verses have the word *son*?," the concordance should be able to do a simple and quick lookup and hand me the answer. Well actually it should hand me a copy of the answer—see the next item.
- Your concordance should *NOT* give out the collection of references it is storing for a given word. If it does so, you can modify the list and affect future searches. In fact, if you notice that when you search for words and phrases several times that the list seems smaller the second time you asked for the same thing, you probably accidentally modified the lists.
- Related to the last point, when you call methods like *retainAll*, *addAll*, *removeAll*, etc., remember that they modify the list you are calling the method on.
- Sometimes when you split strings, the first and/or last string in the returned array is the empty string. You need to be aware of this and make sure you act appropriately. If you are getting no results when there should be results, look into this as a possible cause.
- Finding the phrases between double quotes is one of the trickier parts, so be prepared to do a little (O.K., maybe more than a little) thinking.

- If the input contains an odd number of quotes, ignore the final quote (that is, assume that what is after it is not a phrase).
- Take into account empty quotes (""), extra spaces before, between, and after words, etc. You do not need to worry about extra spaces within a quote. That is a little trickier.
- The following method removes the HTML and other formatting from the verses (mostly for the ESV), removes **'s** at the end of words, and then extracts just the words (ignoring punctuation, etc.) and returns them as an ArrayList. This can be used to extract words from the verses as well as the user's input.

```
public static ArrayList<String> extractWords(String text) {
  text = text.toLowerCase();
  // Removes a few HTML tags (relevant to ESV) and 's at end of words.
  // Replaces them with space so words around them don't get squished
  // together.  Notice the two types of apostrophe—each is used in a
  // different version.
  text = text.replaceAll("(<sup>[,\\w]*?</sup>|'s|’s|&#\\w*;)", " ");
  // Remove commas. This should help us match numbers better.
  text = text.replaceAll(",", "");
  String[] words = text.split("\\W+");
  ArrayList<String> toRet = new ArrayList<String>(Arrays.asList(words));
  toRet.remove("");
  return toRet;
}
```

- Here is a line that will put bold tags around every occurrence of a String *word*, but keep the case of the words intact. This is similar to the one I gave you last time, but this one ensures that only whole words are bolded.

```
text = text.replaceAll("(?i)(?<!\\w)" + word + "(?!\\w)", "<b>$0</b>");
```

  Here are a few details about this regular expression:
  - In Java, the **(?i)** at the beginning of a regular expression is essentially saying "match regardless of case".
  - The **(?<!\\w)** and **(?!\\w)** are saying "there can't be a normal word character here".
  - The < in the first of these is asking the parser to look backwards (this is getting pretty technical, so ignore it if you wish). The reason these are here is that we don't want to bold the "father" in "fatherhood" if fatherhood was the searched word.
  - The **$0** is saying "place here exactly what you matched." Because of the first expression, what was matched was in the original case, so it is replaced with the same thing.
- My Stage 12 tests take about 1.2, 1.1, 1.1, 2.5, 2.6, and 3.2 seconds. If your tests are taking significantly longer than that, you may have serious inefficiencies in your code that you need to locate and remove. There are a lot of places in your code where inefficiencies can creep in on this assignment, so you need to think very carefully about each and every method.
- To determine where your inefficiencies might be, you can do the following:

```
long start = System.currentTimeMillis();
// the code you want to evaluate
long end = System.currentTimeMillis();
System.out.println("Time: "+(end-start));
```

  This will tell you how many milliseconds the code between the two calls to `System.currentTimeMillis` takes.